Available online at www.sciencedirect.com
ScienceDirect
Information Systems 33 (2008) 658–680
www.elsevier.com/locate/infosys

Joining the results of heterogeneous search engines

Daniele Braga, Alessandro Campi*, Stefano Ceri, Alessandro Raffio

Dipartimento di Elettronica e Informazione, Politecnico di Milano, via Ponzio 34/5, Milan, Italy

Abstract

In recent years, while search engines have become more and more powerful, several specialized search engines have been developed for different domains (e.g. library services, services dedicated to specific business sectors, geographic services, and so on). While such services beat generic search engines in their specific domain, they do not enable cross-references; therefore, they are of little use when queries require input from two or more of such services (e.g., "find papers in VLDB 2000 authored by a member of a specified department" or "books sold online written by prolific database authors" or "vegetarian restaurants in the surroundings of San Francisco"). In this paper, we study how to join heterogeneous search engines and get a unique answer that satisfies conjunctive queries, where each query can be routed to a specialized engine. The paper includes both the theoretical framework for stating such problem and the description of pragmatic solutions based on web service technology. We present several algorithms for the efficient computation of join results under several cost model assumptions.

Keywords: Search engines; Web services; Join strategies; Query optimization

1. Introduction

The current evolution of the web is characterized by an increasing number of search engines, ranging from generic ones (such as Google) to domain-specific ones (such as library search systems, or on-line recommendation systems for specific goods, or geo-localization services). While each search engine can be separately used to issue focused queries, their intrinsic limit is the inability to go beyond the purpose for which they have been developed; however, users are often interested in complex queries, ranging over multiple domains. Such queries can be only answered, at the current state of art, by a deep involvement of a knowledgeable user, who inspects search engines one at a time, feeding the results of one search as input to the next one. However, in an ideal scenario, users do not want to be bothered by distinctions between many searching systems and desire to have one common interface available for querying them; moreover, while they can accept a complex interaction when their query is rather complex, they certainly do not want to "cut-and-paste" query results into query inputs, as such approach is time consuming and yields imprecise results.

The focus of this paper is to develop techniques for merging the results extracted from two or more search engines. Our main idea is to design a system offering a common interface to several, known search services such that a user query can be

*Corresponding author. Tel.: +39 0223993644.
E-mail addresses: braga@elet.polimi.it (D. Braga), campi@elet.polimi.it (A. Campi), ceri@elet.polimi.it (S. Ceri), raffio@elet.polimi.it (A. Raffio).

0306-4379/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2008.01.009

decomposed and its keywords can be routed to many of them in parallel. Given that each result is independently built, this approach consists then in *joining partial results into a composite result, to be presented to the user*. The recent availability of web services and XML as a means for generalized interoperability makes such an objective quite feasible.

Joining search engine results is indeed far more complex than joining conventional tables of relational databases: search engine results are ranked lists of complex XML structures which are returned in response to queries, and the pairwise join of the elements of such lists requires specialized optimization strategies. Join is indeed the right underlying paradigm, as we can talk about "join methods" which define the way in which web services are invoked and result lists are examined in order to produce the join results with the best overall performance.

In this paper, we do not address the orthogonal problem of how the user query can be dynamically decomposed and its keywords can be routed to the various search engines: we assume that users are provided with simple interfaces to predefined selections of search services which have been registered in our system, while the dynamic choice of the best specialized service for a given query is outside the scope of this research. Similarly, we do not address the general problem of integrating generic search engine results, which is an instance of the general problem of data integration [1–6], but we assume the existence of ad hoc coupling functions that build the result of joining two given service results by composing given XML elements extracted from them.

1.1. Relevant examples

Our ideal query involves several dimensions, each covered by a given search service. Examples of non-trivial queries that address orthogonal dimensions are the following:

- *Find a good vegetarian restaurant at approximately 30 miles from San Francisco.*
- *Find all VLDB authors from ETH Zurich.*
- *Find pairs of news from the Washington Post and the New York Times dealing with the same event.*

Any human actor recognizes at a glance that these queries can be considered as conjunctions of simpler queries over independent dimensions. The first one requires combining a geographic web service (say, MapPoint) and a recommending service expert in restaurants (say the Michelin Guide, or perhaps a more specific service restricting to vegetarian restaurants); the second one requires combining a web service dealing about publications (say, DBLP) and the staff of a given Faculty, as provided by a given wrapping service over a web site; the third one requires pairing services of two different web sources.

1.2. Technological scenario

The main technological innovation of the last years is the adoption of the service-oriented architecture (SOA); this technology allows applications to exchange information effectively and represents the turning point of several previous approaches to systems interoperability. Several services are already available on the web, not only for specific querying purposes, such as searching book catalogues or dynamically generating maps from geographic information systems, but also for querying general purpose search engines (such as Google) from within other applications. In addition, it is becoming possible to query those information sources which do not expose a web service interface by means of wrappers, which monitor the data contained in such sources and provide a service-based interface for querying the wrapped data [7,8]. This opportunity broadens the scope of the approach described in this paper and opens interesting scenarios for empowering web searches.

This paper discusses the issue of integrating search services so as to answer complex queries by leveraging those (simpler) search services, which are already available. More long-term research concerns web service composition (i.e., the ability to combine web services which are capable of performing tasks so as to build complex workflows) and Semantic Web Services [9,10] (i.e., the ability to select the services that best match a user's goal where both the goal and the services are semantically described by means of knowledge bases or ontologies). Compared to the above long-term research goals, the objective of effectively joining results from different search engines is much easier.

1.3. Approach

Let us consider again the problem of finding authors of VLDB papers of a department. This

would require to query a service giving the staff of the department and a specialized service (or a generic one, if not available) for a list of VLDB papers. Then, results of the two services must be "joined", extracting VLDB papers written by selected authors.

In this paper, we do not focus on how the initial user query can be decomposed into the two subqueries. In the simplest case, a knowledgeable user could select the appropriate services out of a given set of services known to our system, and then interact with each of them by submitting the subqueries. With a more advanced and user-friendly interface, subquery decomposition could be done automatically from a single, initial query, by matching query terms to service descriptions, possibly helped by ontologies; this is the subject of parallel ongoing research.

This paper instead focuses on the join of results produced by two services, and considers such operation as the building block for enabling the use of several services in sequence. The join takes place among ranked lists of XML fragments, where each fragment represents one individual resource descriptor within the search result. The possibility to join the results of two services is decided at the time in which services are registered in the framework. The results of two services are compatible (i.e., "joinable") if a matching function is provided between the schemata of the two web service responses; such schemata are known a priori (from the WSDL description of the services' operations) and the matching is specified in terms of XML query and transformation languages. Of course, the matching may be "stronger" or "weaker", as it may involve more or less restrictive comparisons for different values (e.g., identity, similarity, proximity, compatibility, and so on); also, different matching functions, with different degrees of "strength" (and therefore selectivity) can be defined and registered for one couple of services.

Results returned to users will therefore be based upon the original ranking of fragments and upon the result of matching; presenting the result requires composing the two entries according to a composition strategy which is known for every pair of services. Of course, results are presented to users in "batches", as with conventional search engines, and most batches will not be computed unless the user interacts with the system asking for "more" results. Normally, the join is arrested when relevance is below a given threshold or when enough elements are produced in output; if the user is not satisfied, then the search is continued with the following elements in the relevance ranking. Thus, there is a strong analogy with "top-k" joins of relational databases [11,12]. We extend this approach tackling the challenges posed by the new scenario of results returned by search services.

Also, we require the system to present results to the user in an order that is sufficiently close to their "result ranking", i.e. the ranking that is computed by the system for each result entry. We will see that outputting results in exact ranking order is not convenient, but such ordering can be approximated.

When presenting a new kind of join, it is essential to define metrics for measuring its efficiency and evaluating its cost. In our setting, costs depend on two factors: the interaction with web services through the execution of request–response pairs, and the execution of several XML fragments comparisons after each new web service call. These recall the well-known join cost factors of join optimizers: *i/o costs* (now replaced by request–response pairs) and *cpu costs* (now replaced by the computational costs needed for joining XML fragments, when the matching requires textual comparison and/or domain-specific knowledge). In principle, the two costs adds up; however, we may easily foresee scenarios where the former or the latter cost factor dominates, and we will consider these extreme cases as most representative.

This paper presents a variety of methods for joining two search service. We assume that services are registered and therefore, for each pair of them, the best join method can also be decided at service registration time, depending on the features that are exhibited by each service.

1.3.1. Contributions of the paper

The main contributions of the paper are listed below.

- We state the importance of the problem of joining the results of generic search services for answering cross-domain queries. We propose as well a simple and abstract formalization of the problem.
- We provide a list of examples of queries which are at the same time relevant for the user, very hard or impossible to address with one search service, and feasible with our approach.
- We propose a reference architecture and framework for addressing the join of heterogeneous search services.

- We provide a cost model, several execution strategies, and optimization heuristics for solving the problem, at the highest level of abstraction compatible with its formulation.
- We demonstrate the feasibility of our approach by showing the results we obtained by means of a simple yet complete prototype.

1.4. Paper organization

The paper is organized as follows. Section 2 presents the problem statement, Section 3 is dedicated to join methods, and Section 4 presents examples and experimental results. The paper is concluded with related works and future works.

2. Problem statement

A query $q \subseteq \mathscr{K}$ is a set of keywords taken from an alphabet $\mathscr{K}$. Given n services $S_i, i = 1, \ldots, n$ used for solving q, we assume that q is decomposed into n subsets of keywords, possibly overlapping, such that $q = \bigcup_{i=1}^{n} q_i$, and that each query q_i is evaluated by the corresponding service S_i. The decomposition of q into subqueries q_i is beyond the scope of our research. An example of solution of this problem is [13], that defines a technique TO check whether a query is rewritable within the context of data exchange.

Without loss of generality, we can refer to the case of joins on the outputs of *two* services S_X and S_Y. Operations apply to their responses X and Y and build a result R which is a ranked list of XML fragments built by combining the items from X and Y. The combination of X and Y is not as simple as within the domain of relational databases, as we do not deal with flat tuples, but with possibly complex and nested structures. As we only consider services which expose a web service application interface (either natively or by means of a wrapping layer), XML can be assumed as the common encoding.

As a running example consider the search of restaurants with a peculiar characteristic (say, Greek restaurants) in the surroundings of a given location (say, within 20 miles from Seattle). This task requires to integrate the results collected from a geographic information system for locating places around Seattle, and another service for a list of possible Greek restaurants.

Service S_X: Amazon E-Commerce Service.[1]
Subquery q_X: Greek restaurants.
Service S_Y: GeoPlace.[2]
Subquery q_Y: Seattle; 20 miles.

2.1. Definition of service responses

Given a generic service S and a query q, we denote as X the response returned by S on q; we say that q produces X on S, or, more compactly, $S \triangleright|_q X$. Since all the services S dealt within the paper expose a web service interface, we also assume that X is a sequence of valid XML fragments w.r.t. the schema definitions contained into the WSDL description of the request–response primitive for S. We denote as N_X the cardinality of X, i.e. the number of items in the result.

In our running example X is composed of items describing restaurants, with name, full address, price rating, and other features. *Results* Y: items describing places, with name, zip code, and distance from the center of Seattle.

Each item $x_i \in X$ is provided with a scoring value $\rho_i \in [0, 1]$, which is an estimate of the relevance of x_i w.r.t. the query q. We assume that services return items x_i in scoring order, although they may not return ρ_i itself.

Given that each item x_i is a fragment, we can also think of it as a collection of PCDATA and attribute values, where each such value is defined by a suitable path expression, valid w.r.t. the fragment's schema. Formally,[3] $x_i = (\langle val_{ij} \rangle, \rho_i)$, with $val_{ij} = (path_j, \langle v_{ijk} \rangle)$, where $\langle v_{ijk} \rangle$ is the list of values obtained by the evaluation of the path expression $path_j$ within the XML fragment x_i.

2.2. Join of two search services

The join between two search services requires the definition of a composition operation between

[1] Amazon Web Services are a collection of services offered by www.amazon.com. The Amazon E-Commerce Service provides information about music, movies, books, and other stuff available at Amazon's; in particular, the service presents a small collection of US restaurants.

[2] http://www.codebump.com/services/PlaceLookup.asmx.

[3] Throughout the paper curly braces like $\{e_i\}$ denote a *set* of elements e_i, angle brackets like $\langle e_i \rangle$ denote an *ordered* sequence of *homogeneous* elements, regular brackets like $(e_1, \ldots, e_n)$ denote a *n-uple* of *heterogeneous* elements, and square brackets like $[v_1, v_2]$ denote an *interval* between two values.

items, which is based on a matching test and computes a ranking value for the couple.

Given two services S_X and S_Y such that $S_X \triangleright|_{q_X} X$ and $S_Y \triangleright|_{q_Y} Y$, and borrowing a notation from classical relational algebra, the query $q = q_X \bowtie q_Y$, we write $R = X \bowtie Y$ (as usual, $\bowtie$ is the join operator). R is a sequence of elements $r_k = \langle c(x_i, y_j), \rho_k \rangle$ which answer the original query q and such that x_i and y_j are selected items taken from X and Y, respectively; $c(\cdot)$ denotes a *composition function* to be properly applied to the components of x_i and y_j, which is set up in the service registration phase; and the term ρ_k represents the ranking of $c(x_i, y_j)$, i.e., an estimate of the relevance of r_k with respect to q.

As for the match to be performed by the composition function, it requires (i) identifying pairs of XML elements to be compared and (ii) choosing a matching operator:

- *Choice of XML elements.* We assume that a comparison matrix C_{XY} is available for each couple of services S_X, S_Y, telling which pairs $(path_h(x_i), path_k(y_j))$ need to be compared in order to compute ρ_k. The higher is the weight contained in cell c_{hk}, the higher is the importance of the comparison between values extracted by $path_h$ on X and by $path_k$ on Y. Note that the comparison matrix C_{XX} of a service with itself is the identity matrix.
- *Choice of matching operator.* The matching between pairs of XML values (elements or attributes) can take place using several different operators; such operators can be organized in a hierarchy, rooted on (typed) equality, that includes text containment, partial textual matching, and possibly ontology-driven term similarity. Clearly, when comparing two strings, (e.g., the "location" of a restaurant), their identity yields to a higher matching index than partial string matching or term similarity. It is possible to evaluate each pair with a different operator. The estimation of ρ_k requires comparing the two elements x_i and y_j according to the given *matching strategy*. The result of the match operation is a match index $m_{ij} \in [1, 0]$, giving the correlation between the two elements x_i and y_j (how much they describe the same resource or real world fact or event). Note that match indexes have no counterpart in the relational join, as two values being compared are either identical ($m_{ij} = 1$) or different ($m_{ij} = 0$). Then, the relevance ρ_k is simply given by the product[4] of the two scoring orders of the fragments x_i and y_j, produced by the search engines, and the match index, produced by the matcher:

  $$\rho_k = \rho_i \times \rho_j \times m_{ij}.$$

  Note that every term in the right side of the above formula ranges in the $[0, 1]$ interval and therefore also $\rho_k \in [0, 1]$.

We can think of many different ways to register a service within our framework, that lead to different techniques for obtaining the aforementioned comparison matrix C, ranging from naive direct coupling performed by human intervention, feasible when the framework deals with few predefined services, to sophisticated semantic-aware techniques, which are useful to build a flexible system capable of addressing many different services, dynamically identified, and coupled. The determination of C goes beyond the scope of our research; we simply assume the matrix C_{XY} as available in the environment, containing the weights c_{hk} and the indication of the matching operator to be applied.

For the query of the running example, $c(\cdot)$ is quite simple: only two path expressions are involved (the zip code of towns provided by GeoPlace, in the element tagged *toPlace*, and the zip code in the address of a restaurant provided by Amazon, in the attribute named *postalCode*). The matching method is strict string equality, thus m_{ij} is 1 if the strings match, 0 otherwise. Fig. 1 shows one of these results, where *postalCode* from Amazon equals *toPlace* from GeoPlace.

In the running example, the Amazon service returns a list of restaurants; ranking values ρ_{x_i} for this list can be approximated by a constant function, since the service returns only the restaurants that match the given query (they either serve Greek food, or they do not). For the GeoPlace service the rank ρ_{y_j} of each place is easily computed by means of a distance function, giving score 1 to central places and 0 to places 10 miles away.

We describe in Section 4 an architecture for registering search services, that shows how these ingredients can be implemented.

[4]We assume for simplicity that the combination of such values is done by multiplication. Of course any other triangular norm can be applied.

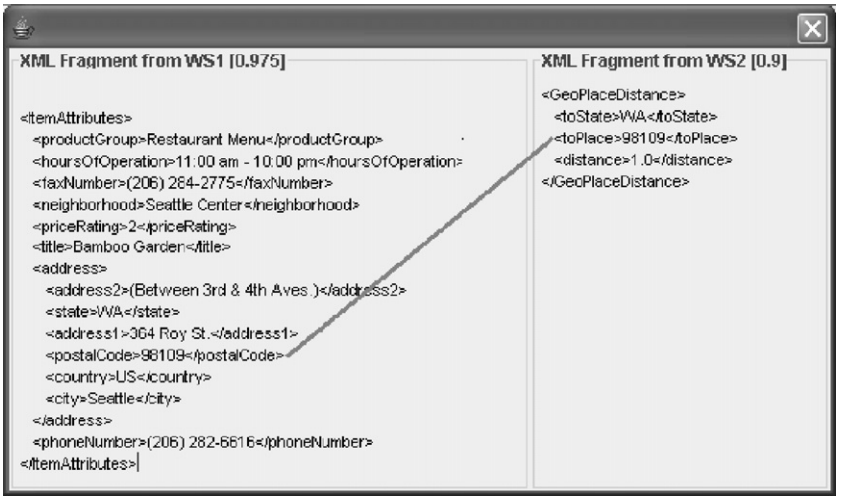

Fig. 1. Greek restaurants near Seattle (matching fragments).

2.3. Efficiency measures for optimization

In order to prepare the background for addressing optimization issues, we first need a notion of optimality for join strategies. Given the definitions above, it is possible to define an abstract notion of *optimality*: given a query q and a threshold $\overline{\rho}$ such that $1 \geqslant \overline{\rho} \geqslant 0$, a join strategy is optimal, relative to $\overline{\rho}$, if it produces output elements $r_k = \langle c(x_i, y_j), \rho_k \rangle$ such that $\rho_k \geqslant \overline{\rho}$, in exact ranking order, with the minimum cost. The cost depends on the adoption of a specific cost model, whose factors include the cost of interacting with services and the cost of computing the matching between service results. Of course, if the strategy achieves optimality, then the system can also present results in ranking order, i.e. with monotonically decreasing values of ρ descending from 1 to $\overline{\rho}$.

However, this notion of optimality is neither precise enough nor practically desired. First, it does not define the matching test producing ρ_k, and of course such test can be arbitrarily complex by itself (it could, for instance, require the intervention of external ontologies for term matching, possibly available through web service calls, hence involve comparable costs as interacting with the engines). Second, it may be inappropriate to produce results in strictly decreasing ρ values, because to achieve such result it could be required halting the output production for each result entry until the system decides that no higher ranking result entry will ever be produced. So we introduce a revised notion of optimality, that does not depend on the matching, and that does not achieve a perfect ranking of results, but only an approximate ranking.

We say that, given a query q and a threshold $\overline{\rho}$ such that $1 \geqslant \overline{\rho} \geqslant 0$, a join strategy is *extraction-optimal*, relative to $\overline{\rho}$, if it produces all elements $r_k = \langle c(x_i, y_j), \rho_k \rangle$ such that $\rho_k \geqslant \overline{\rho}$ in decreasing order of the product of the two rankings $\rho_i \times \rho_j$ and with the minimum cost. Extraction-optimality guarantees that a result entry whose (computed) ranking falls below a given threshold will not be presented, but then presents results above threshold in the order in which they are extracted from the services; therefore, the dataflow of results produced to the user is not "blocking" (by abusing of the terminology which is typical of query streams), and results can be "batched" to users while they are extracted from the search engines.

3. Join strategies

As mentioned above, we assume that examined services return results in decreasing ranking order. Such ranking can be observed while the elements are being retrieved one "batch" (or block) at a time, by reading their ranking values.

In order for the final results to have the chance of being collected in relevance order with respect to the query, the ranking values of the results coming from each service S_X must be function of the subset of keywords used for that service. This hypothesis is necessary to assert that good results can be obtained as a combination of the topmost answers of the involved services. For example, suppose that we are looking for restaurants in the surroundings of a given town, ordered by distance. This would be unfeasible if the towns returned by the geographic

service were ordered alphabetically, rather than by distance.

We can represent the blocks of the results extracted from two services S_X and S_Y over the axes of a Cartesian plan, such that on each axis the ranking order of the blocks decreases from the origin down to the end of the list (see Fig. 2). Each point P in the plan represents a couple (x_i, y_j) which is a candidate $r_k \in R = X \bowtie Y$, where $S_X \triangleright|_{q_x} X$ and $S_Y \triangleright|_{q_y} Y$. We divide X and Y in blocks named b_{Xi} and b_{Yj} respectively, where b_{Xi} is returned by S_X in response to its ith call and b_{Yj} is returned by S_Y in response to its jth call (indicated $S_X \triangleright|_{q_{x\,i}} b_{Xi}$ and $S_Y \triangleright|_{q_{y\,j}} b_{Yj}$). The Cartesian plan is thus divided into rectangles of size $n_X \times n_Y$, where n_X and n_Y represent the size of the blocks expressed as the number of retrieved items. We call *tile* t_{ij} the rectangular region that contains the points relative to blocks b_{Xi} and b_{Yj}. Two tiles are said to be *adjacent* if they have one edge in common.

The plan is a model of the search space to be explored by the joiner. Each rectangular region of size $m \times n$ represents the part of the search space that can be inspected after performing m request–responses to S_X and n request–responses to S_Y. Therefore, achieving extraction-optimality requires a suitable exploration strategy for such search space, which guides a "careful scan" of the result lists.

3.1. Scenarios

Concerning the cost model, we consider the following scenarios:

A. *Cost of request–response execution dominated by join execution*. We assume that joining the blocks that become progressively available is the expensive operation, e.g. because each element comparison requires itself a call to a semantic web service.

B. *Cost of join execution dominated by request–response execution*. We assume that once a block of XML elements is retrieved as the effect of a request–response to services, then join requires simple main-memory comparison operations and can be neglected.

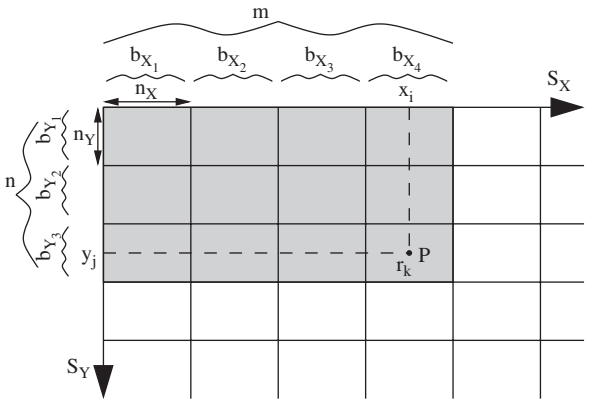

Fig. 2. Search space in bi-dimensional representation.

Concerning the availability of rankings, we start by assuming that ranking be available for both services as one of the parameters which are returned together with each XML entry. However, ranking can be unavailable on either one or both services. In this case, we further characterize the service according to its expected behavior in the following two classes:

1. *Step ranking*. We assume that, by performing a limited number K of request–response, most of the relevant entries can be retrieved, because the entry relevance decreases with a step, and such deep step is normally captured by doing K request–responses. For simplicity, we assume K to be a parameter associated with the service.
2. *Linear ranking*. We assume that the entry relevance decreases roughly linearly, with no step.

Note that the unavailability of the ranking function to the planner does not affect our basic assumption that the search services return their result in ranking order, but simply describes the situation in which this function is opaque to the optimizer.

Of course, cases A/B and 1/2 represent two neatly different behaviors; they are defined so as to enable us discussing particular techniques which would be convenient if they occur, and then performing experiments with actual services so as to check experimentally which technique better responds to the specific choice of web services. In the next section we then define cases A and B, and then the variations A1, A2, B1, and B2 when rankings are unavailable under the two cost assumptions.

3.2. Dominance of matching costs

When matching cost dominates (case A according to Section 3.1), the best strategy is to choose at each step the tile with highest "ranking", i.e., the tile with the highest potential of producing the best results. The estimation of tile rankings is discussed next.

3.2.1. Tile extraction optimal (TEO) method

As we are interested in exploring the most promising tiles first, we define an estimate a_{ij} of the expected relevance for the results of the join operation performed on the points of tile t_{ij}. For simplicity, we use the first point of each tile (i.e., the one with smaller coordinates) as representative for the tile. We also use the ranking ρ of the *last* element returned by each service call as an estimate of the ranking of the *first* element that would be returned by the next call to that service (as a consequence of the hypothesis stating that services return results in ranking order, even if the ranking value is hidden).

All exploration algorithms begin by initializing two sets, $\mathscr{E} = \emptyset$ and $\mathscr{U}$ initially containing all possible tiles, representing the *explored* and *unexplored* tiles, respectively. During the exploration, tiles progressively migrate from $\mathscr{U}$ to $\mathscr{E}$, until the end of the search.

We also define a set $\mathscr{C} \subseteq \mathscr{U}$ of *candidates*: tiles in $\mathscr{C}$ are all tiles t_{ij} which are adjacent to some tile in $\mathscr{E}$. Further, a tile in $\mathscr{C}$ is *dominated* by another tile of the same set when it is adjacent to another tile in $\mathscr{C}$ with smaller sum $i + j$, where i and j are the indexes that locate the tile inside the search space.

Due to the assumed monotonicity of the relevance estimates of the service responses along each axis, dominated tiles have a lower expected relevance than their respective dominating tiles. Fig. 3 shows four possible configurations of the Cartesian plan, where tiles in $\mathscr{E}$ are represented in black, tiles in $\mathscr{U}$ are represented in white, and tiles in $\mathscr{C}$ which are not dominated are marked with a dot.

Referring to Fig. 3, we can equivalently say that (a) each column or row contains at most one dotted tile and (b) for each dotted tile all tiles whose indexes are (both) lower or equal belong to $\mathscr{E}$ and all tiles whose indexes are (both) higher or equal belong to $\mathscr{U}$. Therefore, after $n + m$ request–response operations, it is guaranteed that the cardinality of the set of nondominated candidates is between 2 and $\min(m, n) + 1$.

Given the notion of dominance between tiles as defined above, the algorithm tile extraction optimal (TEO) is presented in the following algorithm.

Algorithm TEO.

1. Compute t_{11} (this requires performing an initial request–response to S_X and S_Y, retrieving b_{X1} and b_{Y1}).
2. Compute $\mathscr{E}$, $\mathscr{U}$, and $\mathscr{C}$;
3. Compute the estimate of rankings ρ for all nondominated tiles in $\mathscr{C}$.
4. Choose in $\mathscr{C}$ the tile t_{ij} with the highest estimate for the average expected relevance a_{ij};
5. If t_{ij} has either i or j set to 1, then perform a request–response to the relevant service.
6. Perform the join operation over all the items in t_{ij} by applying the composition function $c(\cdot)$.
7. Output those results which are above the relevance threshold.
8. Goto step 2 unless (a) the search space is exhausted or (b) the user stops the search.

The algorithm is TEO because it extracts tiles in decreasing order of the tile's estimated rankings.[5] Strictly speaking, extraction-optimality as defined in Section 2.3 would require operating *element by element*, by "anticipating" request–responses, i.e. performing two initial request–responses in step (1), and then building the joins in all nondominated elements of $\mathscr{C}$, outputting results in decreasing ranking of $\rho_i \times \rho_j$. However, the proposed algorithm is indeed much simpler and is extraction-optimal under the two simplifying assumptions of selectivity estimates.

3.3. Dominance of service requests

In this scenario (case B according to Section 3.1), we assume that request–response execution dominates over matching costs; therefore, the most efficient approach is to compare all the elements in all the retrieved tiles in all the possible combinations before submitting another service call.

3.3.1. R-shape method

As the cost of joining *all* the pairs in a tile is dominated by the cost of performing *one* single i/o operation, the choice reduces to choosing at each step (i.e. at each request–response) which service is to be invoked next. After retrieving each batch of new results, the join is performed between the new items and all the cached items, thus exploring an entire row (or column) of tiles.

Note that after K request–response operations performed over service S_X, one call to S_Y allows the joiner to explore K new tiles. Fig. 4A shows by

[5] We recall that we estimate the ranking of all the elements of a tile as its first element and that we use, for candidate tiles requesting a new web service call, the ranking of the last element retrieved by the last web service call.

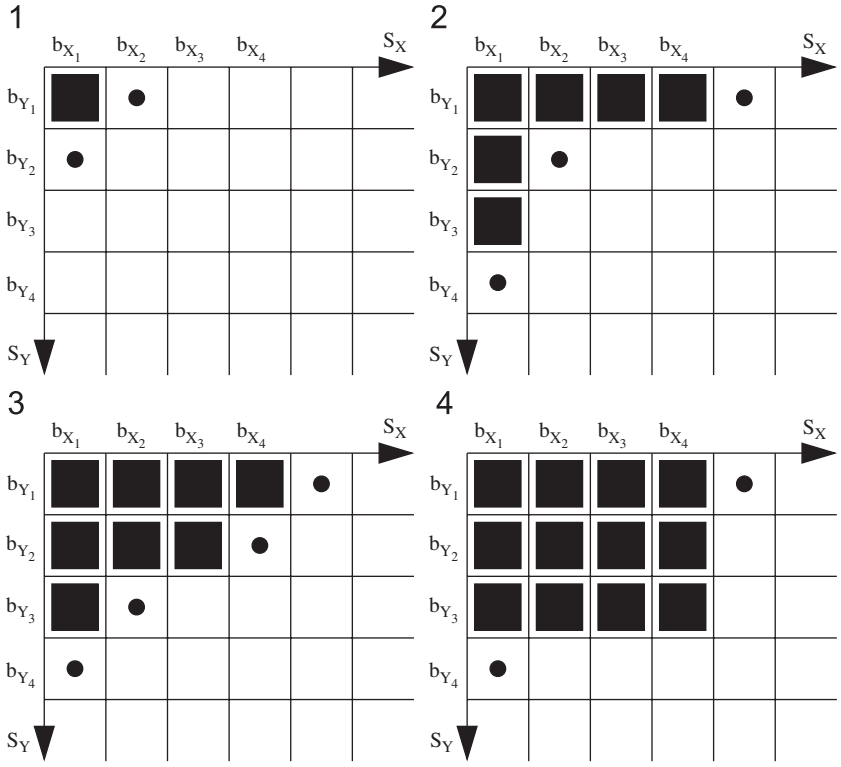

Fig. 3. Deciding the next step in the scan.

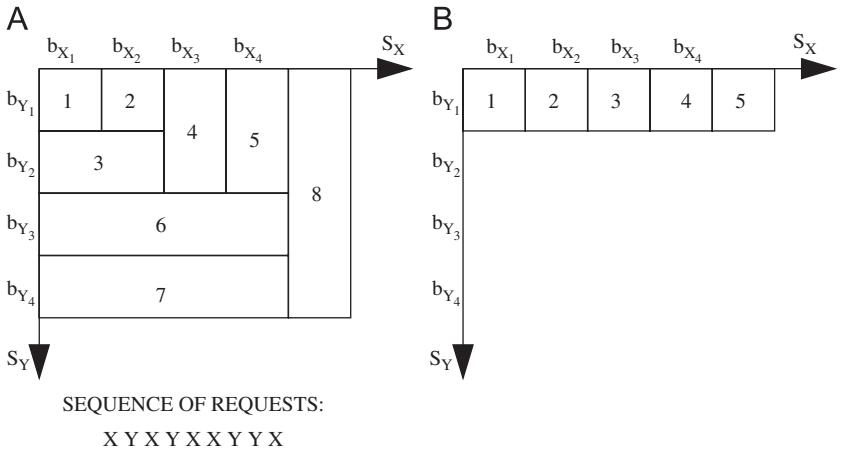

Fig. 4. R-shape.

means of numbering from 1 to 8 the order in which the first eight regions of the search space are explored, in a case in which the ranking trends suggest that the most convenient schedule for the first eight service calls is the following: S_X, S_X, S_Y, S_X, S_Y, S_X, S_X, S_Y, S_Y, and S_X.

It should be noted that a strong asymmetry in the ranking of the two services may lead to a "long and thin" rectangular area composed of the already explored tiles. This degenerates, in the worst case, to addressing all the calls to one service only (except for the first two calls, which are always alternated so

as to have at least one tile for starting the exploration). This particular case, shown in Fig. 4B has the disadvantage that each i/o only adds one tile; to avoid this, one may want to (heuristically) call a few batches on both services before applying the algorithm. Some further heuristic opportunities for trying to anticipate the retrieval of tiles which are likely to produce good matches will be illustrated in Section 4.4.

3.4. Join strategies in the lack of rankings

While in the previous sections we assumed that search services would return results containing explicit ranking values, in many real-life situations search services return results with opaque ranking. In these cases, due to the lacking of an explicit ρ, neither the extraction-optimal strategy nor the R-Shape strategy are applicable. Therefore, one has to resort to simpler join strategies, and we consider two extreme cases, corresponding to two possible ranking trends:

- The service returns by results which satisfy the query up to a certain entry, and therefore the opaque ρ has a trend which can be approximated by a step; we also assume to be able of guessing where the ranking briskly decreases (i.e., estimating the result size).
- The service returns results which are progressively less relevant to the query, and therefore we assume a progressive decrease of ρ, whose trend is approximated by a linear slope.

This classification arises four possible cases for the join of two services:

- The three cases in which at least one of the two trends has a step are approached by three variations of a strategy named *nested loop*.
- The remaining case (that of two linear slopes) has to be approached in two different ways, depending on the dominant cost; we will first consider the matching costs as dominant, addressed by a strategy named *merge-scan*, and then the service call costs as dominant, addressed by a strategy named *S-shape*.

Given that the features of each service are known in advance, these strategies can be decided at service registration time. The overall approach is visually summarized in the following table, while the strategies are described next.

		Trend of S_X Step	Slope
Trend of S_Y	Step	*Nested loop* (the inner service is the one with smaller expected result size)	*Nested loop* (Y is the inner service)
	Slope	*Nested loop* (X is the inner service)	*Merge-scan* *S-shape*

3.4.1. Nested loop

The nested loop strategy is suitable when the results of one search engine exhibits a clear "step"; in such case, we assume that the ranking suddenly drops from a high value to a very low value. The corresponding best exploration strategy of the search space reminds of the "nested loop" method for relational joins; the exploration consists of extracting all the blocks corresponding to the high ranking values of the "step" engine, and then extracting the blocks of the other service in ranking order, thereby producing join results. This strategy is represented in Fig. 5.

3.4.2. Merge-scan

The merge-scan strategy is indicated in the absence of information about a clear "step" in the ranking of results. Then, one should assume that rankings decrease linearly. The corresponding best exploration strategy of the search space reminds of the "merge-scan" method for relational joins; the exploration consists in moving "diagonally" in the Cartesian plan, as shown in Fig. 6, where the arrows indicate the order in which the tiles are chosen starting from t_{11}.

3.4.3. S-shape

The square shape strategy translates the merge-scan case to the situation where matching costs are negligible; therefore, as in the previous case, web services are called in alternation; however, the entire set of tiles (now building a square rather than rectangular shape) are considered. Fig. 7 shows this join strategy.

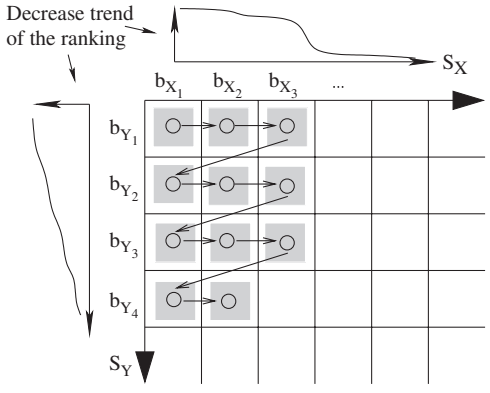

Fig. 5. Nested loop.

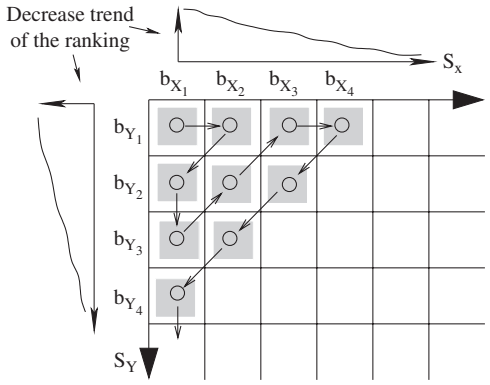

Fig. 6. Merge-scan.

4. Experimental evaluation

The research described in this paper is part of a project dedicated to the development of a *search engine integration framework*. We initially present the overall architecture and then the prototype that we have implemented so far, with which we have conducted the experiments described in Section 4.2.

4.1. Joiner architecture

Fig. 8 shows the main components of the search engine integration framework:

- The *query user interface* takes as input queries from the user in the format of collections of terms and shows the results as output; with an interactive adaptation, it allows users to indicate their browsing preferences within the results through simple interactions.
- The *query decomposer* reduces the original query into several (possibly overlapping) subqueries and identifies the services that can better address each subquery, using the information about registered search services coming from the *search service directory*.
- The *joiner* selects the join method, and whenever appropriate interacts with the service caller in order to request a new block of results from a search service.
- The *matcher* performs the join of the elements in a given tile, producing the result entries; the choice of the matching algorithm depends on the correlation of each pair of registered services, as provided by the *directory service*.
- The *composer* builds the entry results which are sent to the *user interface* for their publishing.
- The *ranking guesser* is an optional module that estimates the average ranking of each block of records returned by a search engine call when such number is not explicitly provided as a parameter of the response from the service.
- The *service caller* is activated by the *joiner* to request a new block from a given search service.
- The *registration service* is used to register web services within our framework. This is essentially a manual operation: the WSDL of the service is analyzed, and the correlation of the new service with respect to all the other services is derived and specified; in addition, other features of the service are registered, such as the typical trend (linear vs step) of the ranking function, the average web service performance (i.e., the delay required by a request–response pair) and the average number of blocks returned by each web service call. This information is stored in the directory service.

Our prototype is limited to the modules surrounded by the dot-dashed line in Fig. 8. More precisely, our prototype takes as input two queries, a specification of the composition function $c(\cdot)$ suitable for matching two specific services, and additional information about the two services (such as WSDL descriptions and other meta-data); with such input, the tool performs service calls, joins the partial results according to one of the strategies described in Section 3, and outputs the combined results in a stream, as soon as they are ready (in extraction order). The initial query is decomposed manually into subqueries and translated to fit the specific service interface; all aspects concerning the user interface and the registration of web services in some ontologies, which would enable

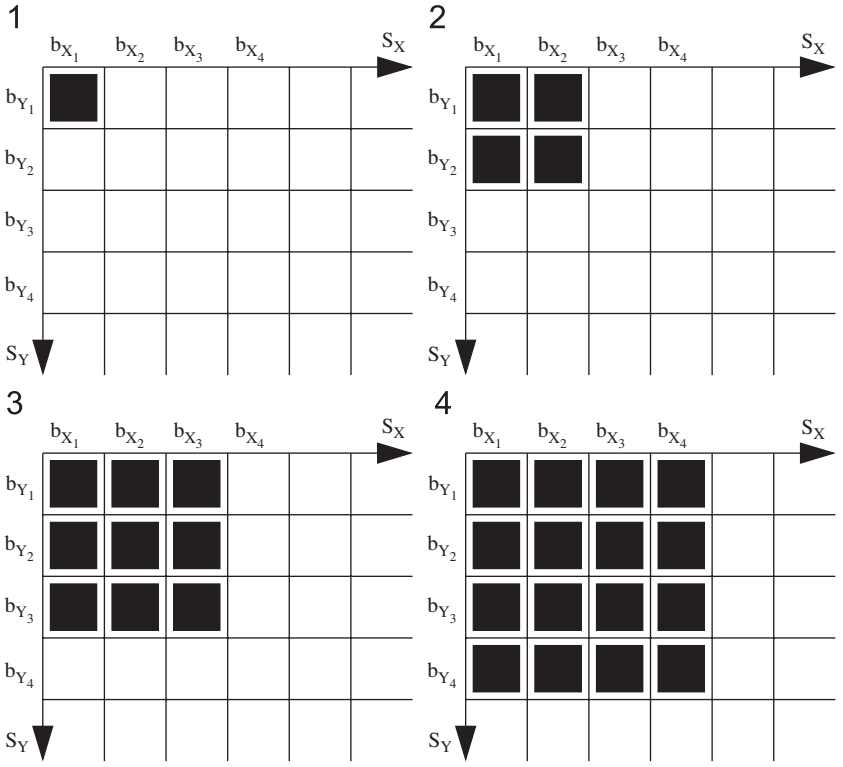

Fig. 7. S-shape.

the semi-automatic creation of the composition function $c(\cdot)$, as well as the decomposition of queries and redirection of keywords to the proper service, will be addressed by future work.

4.2. Experiments

Experiments were conducted on a Core 2 1.8 GHz with 2 GB of memory and 300 GB of disk space; for the purposes of this work, which studies the feasibility of the approach, local buffer space can be considered unlimited. The issue of resource boundaries in large environments will be addressed by future work.

The prototype has been implemented in Java 5.0; in particular, the interaction with web services builds on the Apache Axis framework.

We conducted a series of experiments, using some popular search services available on the web. Our experiments compare the various join methods by plotting the number of service calls and the number of explored tiles as functions of the number of retrieved "good" results (defined as such by knowledgeable users). This kind of visualization allows one to rapidly recognize how the "cost" of interesting results grows with the number of the results themselves, as well as to easily compare the "extraction optimality" of different strategies. Indeed, this allows one to compare the cost of retrieving the "first k" good results, expressed in terms of the dominant cost factor (in particular, we are interested in $k \leqslant 10$, so as to show fast the first page of results). We considered the methods discussed in the previous section within the context of both cost models described in Section 3.1. Of course, different strategies do not only differ in the cost of extracting the first k results, but also for the order in which the results are returned.

4.2.1. Typical restaurants close to a given city

Our first use case is concerned with the search of restaurants within a given area. We start with the running example presented in Section 2: find Greek restaurants within 20 miles from Seattle, WA.

Service S_X: Amazon E-Commerce Service[6]

Subquery q_X: Greek restaurants.

[6]Amazon web services are a collection of services offered by www.amazon.com. The Amazon E-Commerce Service provides information about music, movies, books, and other information

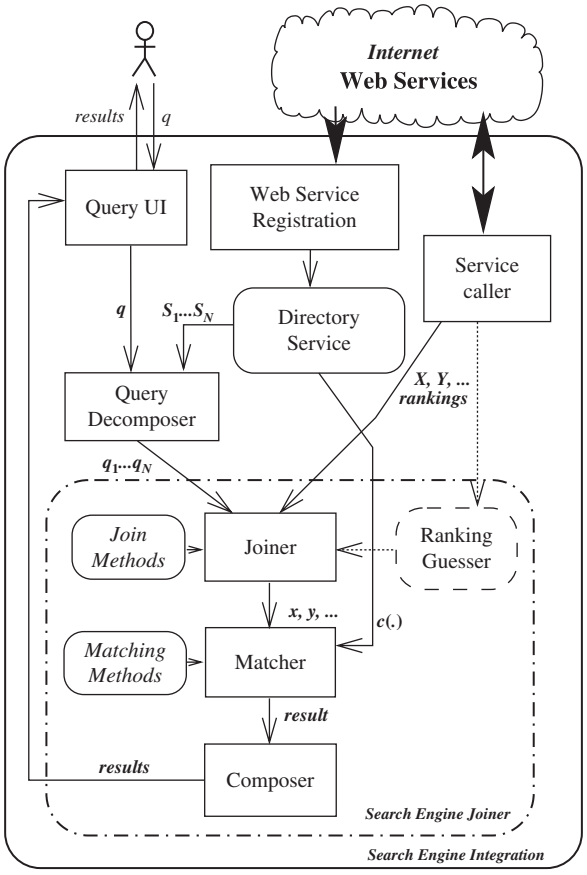

Fig. 8. Architectural view of the overall framework. Our prototype covers the modules surrounded by the dot-dashed line.

Results X: Sixty-three items, in blocks of ten per request. Each item describes a restaurant, with name, full address, price rating, and other features.[7] The ranking function for this list can be approximated as a step, since the service only returns the restaurants that match the given query (a restaurant is either Greek or not).

Service S_Y: GeoPlace[8]

Subquery q_Y: Seattle, WA; 20 miles.

Results Y: Fifty-eight items, in blocks of ten per request, ordered by increasing distance. Each element describes a place, with its name, zip code and distance from the center of Seattle. The rank of each place is easily computed by means of a distance function, giving score 1 to central places and 0 to places 20 miles away.

Composition function: For this query, the matching method $c(\cdot)$ is plain string equality between the two path expressions representing zip codes.

Good results: Manual analysis of results found five interesting Greek restaurants located near Seattle.

Considerations: This example corresponds to the B scenario (as classified in Section 3.1), where the cost is dominated by the interaction with the remote services: indeed, the join operation is trivial. The ranking trend of the Amazon service presents a step, while the GeoPlace service's ranking gradually decreases. Given these trends, we expect the nested loop method to be the most appropriate (with Amazon as the inner service). The method matches each place, starting from the one closest to the center, with all the restaurants; in this way matching restaurants are progressively extracted in a proximity decreasing order. Fig. 9 confirms our intuition, as the nested loop method gets all the matching restaurants by means of fewer invocations of the services. The method is compared to the merge-scan, to the S-shape, and to the alternative nested loop (i.e., comparison of each restaurant with every place). The search space is represented by the table shown next to the diagrams. Note that all the methods converge to the same final result of retrieving the same number of restaurants. The figure shows that the other strategies, which perform worse on the whole result set, return their first three results earlier than nested loop; however, it has to be noted that they first retrieve two restaurants which are far from the center, thus achieving a worse approximation of extraction optimality.

Similar cases: Figs. 10 and 11 show the results of similar queries on the same two services.[9]

In particular, Fig. 10 is about vegetarian restaurants within 20 miles form San Francisco, CA. Sixteen places out of 70 matched the conditions, and the nested loop strategy (with results from Amazon in the inner loop) overall performs far better than the other ones. Note that merge-scan also behaves rather well at the beginning, and that S-shape performs well on the overall set, but again nested-loop also better achieves extraction optimality.

(*footnote continued*)
available at Amazon's; in particular, the service presents a small collection of US restaurants.

[7]Data schemas of this and other services are omitted for brevity.

[8]http://www.codebump.com/services/PlaceLookup.asmx.

[9]Since the dominant cost of this scenario is that of request/responses, these figures only compare the hits against the request/responses needed for each strategy.

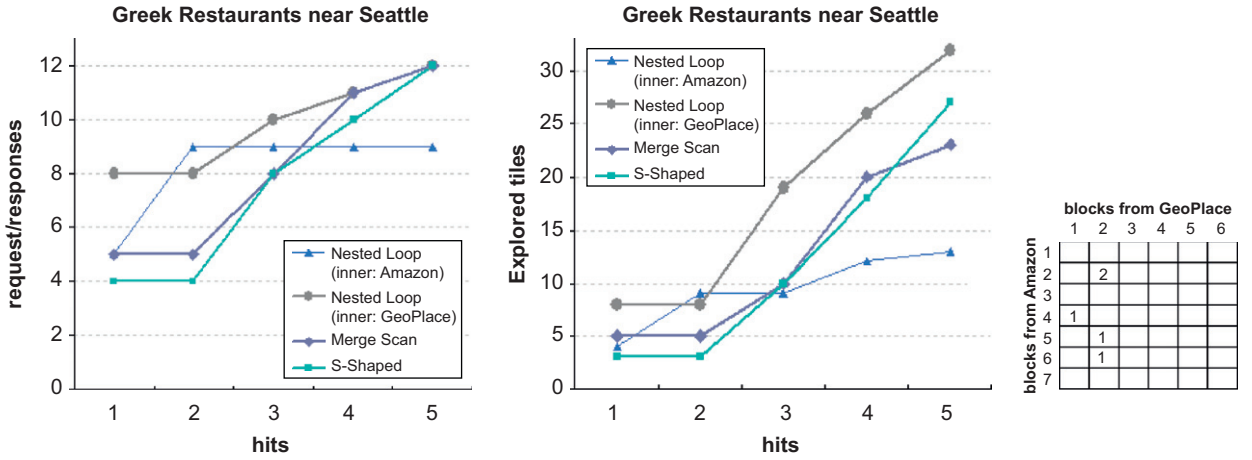

Fig. 9. Greek Restaurants in Seattle vs request/responses and tiles. The graphs show the cost of obtaining a certain number of hits with respect to both cost factors. Due to the specificity of the use case, the dominant cost is that of request/responses. The table represents the search space and shows the number of hits in each tile.

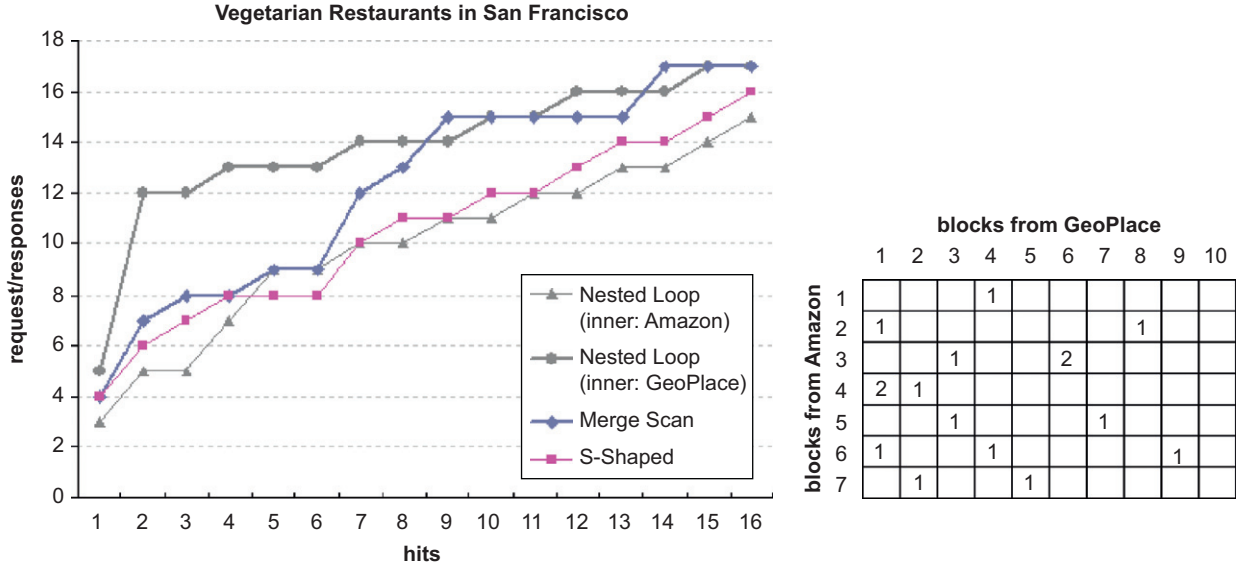

Fig. 10. Vegetarian restaurants in San Francisco.

Fig. 11 shows the results that we obtained asking for pizza in San Francisco, where six restaurants out of 98 matched the join criterion against the 45 places returned by GeoPlace.

In summary, we can conclude that nested loop is the strategy that dominates the three examples, not only in terms of cost but also in term of extraction-optimality, as it best adapts to the decreasing trend of the ranking based on the distance from the town center.

4.2.2. Retrieving books

This use case is concerned with finding books authored by researchers who are "specialized" in the book's subject (something related to databases), i.e., researchers who have also published papers on that topic.

Service S_X: Amazon E-Commerce Service.

Subquery q_X: Books about databases.

Results X: More than 60 K results were returned, in blocks of 10 per request. For our experiments, we

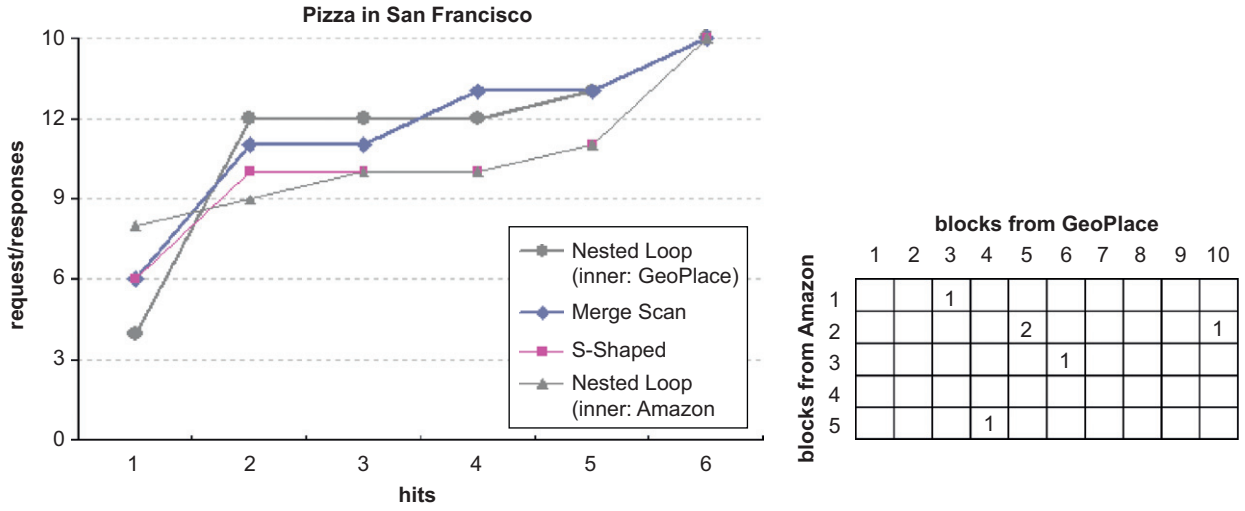

Fig. 11. Pizza in San Francisco.

stopped after 14 blocks. Each item describes a book, with ISBN code, title, list of authors, publication year, and other features.

Service S_Y: The DBLP repository[10]

Subquery q_Y: List of articles from VLDB.

Results Y: In order to reduce the search space, only 100 articles were retrieved. Each item describes an article, with title, list of authors, and more.

Composition function: For this query, $c(\cdot)$ is quite expensive: the service composition function needs to (i) check the equality of the authors and (ii) check the semantic similarity between the title of a book and the title of a paper. While the former operation is quite inexpensive, the latter is performed by means of calls to a web service which provides semantic mappings between terms; for this operation we relied on *MoMiS* [14], a semantic schema mapper developed by the University of Modena.

Good results: Knowledgeable users selected fifteen books whose authors wrote VLDB papers with matching topic.

Considerations: Due to the fact that many calls to an external service occur within the join operation, the cost model for this example is that of scenario A (as classified in Section 3.1): the cost of executing request/responses is dominated by that of the join. Since the ranking functions of the results are opaque for this query, the best approach is to perform a merge-scan navigation. Fig. 12 shows the results of the experiment.[11] Both nested loops strategies are clearly not convenient. S-shape gains at the end due to the lack of hits in columns corresponding to the last two blocks returned by DBLP.

As for testing the TEO and R-shape strategies, which take advantage of explicit rankings, we had to cope with the fact that, unfortunately, conventional search services only provide opaque ranking values. Thus, in order to conduct the experiments described next, we conjectured the rankings deriving the criteria according to the user queries. We calculated the ranking factors by inspecting the XML fragments, so as to field-test TEO or R-shape together with the other approaches. More specifically to this use case, we have manually ranked "recent" publications according to their publication years and "cheap" books according to their price.

We now address the issue of extracting *cheap* books by *currently* active authoritative researchers (i.e., authors of recent VLDB papers).

Service S_X: Amazon E-Commerce Service.

Subquery q_X: Books about databases, now ordered by (ascending) price.

Results X: Same as previous experiment, now ordered by price.

Service S_Y: The DBLP repository

Subquery q_Y: List of articles from VLDB.

[10]The DBLP online service only allows retrieving the whole archive as a single XML document; we have wrapped the service so as to provide basic query capabilities, according to what anticipated in Section 1.2.

[11]Since the joins dominate the cost, the number of visited tiles per hit are presented.

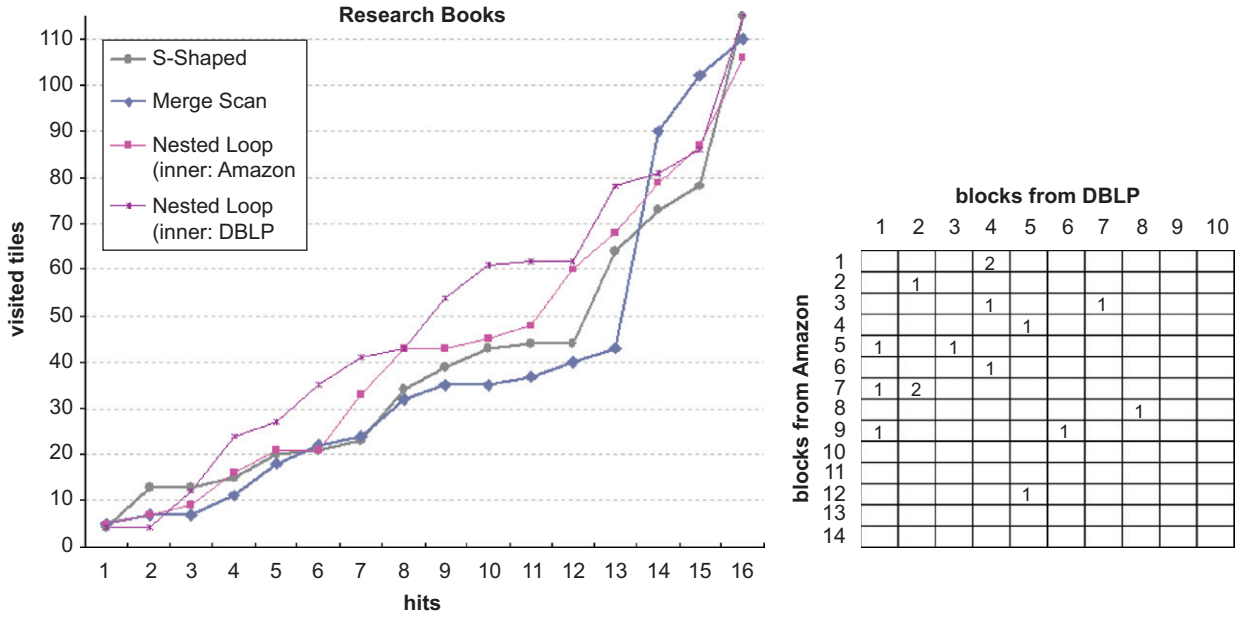

Fig. 12. Books by authoritative researchers.

Results Y: Same as previous experiment, now ordered by publication year, descending (from year 2005 down to year 2000).

Considerations: Both services return ranked results and the scenario is the one in which the join cost dominates. The exploration of the search space can be therefore driven by the expected ranking values, conjectured by the scores assigned to the results of both services. The candidate best strategy is that of the TEO algorithm.

Fig. 13 shows the results of this experiment. The exploration performed by TEO first explores the tiles relative to very cheap books and very recent articles. In order to give an intuition of the strategy in action on this example, the search space shown in the table beside the diagram is marked with the tiles explored by TEO at the time of the 9th hit. Indeed TEO confirms the expectations and proves to outperform the other strategies.

4.3. An anomaly

We now consider a task for which the theory discussed so far did not fit, and on which we discuss further optimization opportunities. The task is similar to the previous one but for the ranking of books. The query asks for recent books about databases (ordered by publication year) written by authors who also recently published at VLDB.

Service S_X: Amazon E-Commerce Service.

Subquery q_X: Books about databases.

Results X: Same as previous experiment, now ordered by publication year, descending.

Service S_Y: The DBLP repository

Subquery q_Y: List of articles from VLDB.

Results Y: As in the previous experiments, VLDB papers ordered by publication year, descending (from year 2005 down to year 2000).

Considerations: Both Amazon and DBLP return results; the navigation can be driven by the TEO algorithm.

If we consider the hits distribution that we found in this case, which is shown by the table in Fig. 14, we note that 15 books match the criteria, and also that many of them are clustered in a specific region. This case is peculiar because of the presence of a particular author who wrote many books on a particular subject. The "anomaly" is the presence of an element (an author) in one of the results of the services which is per se hardly distinguishable from other "near" authors but, differently from them, causes several hits with a high global relevance, as he is the author of many books of interest. This outcome (the high relevance of some pairs) could not be inferred from the estimated joint relevance, which is based on the relevance with respect to the specific services; in fact, even if the TEO algorithm beats a vertical nested loop by a factor of 2, retrieving all those clustered results still requires visiting a huge number of tiles. Indeed, results are

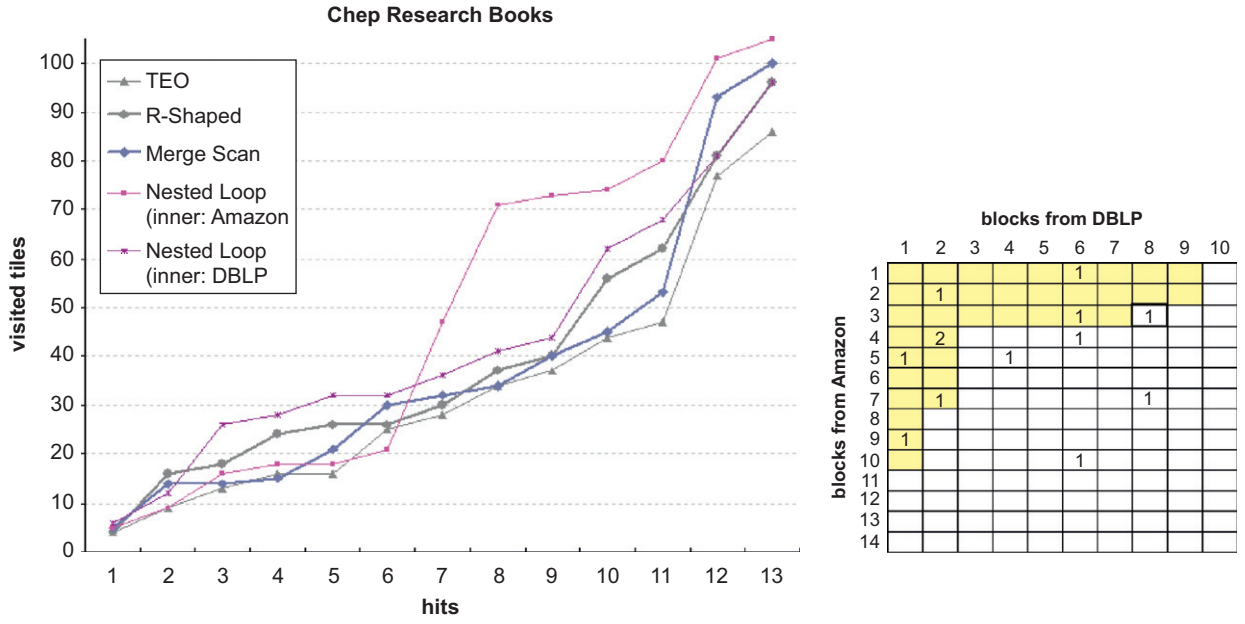

Fig. 13. Cheap books about databases by currently active VLDB authors. In the search space, the 36 marked tiles are those already explored by the TEO algorithm when the 9th hit occurs on the framed tile.

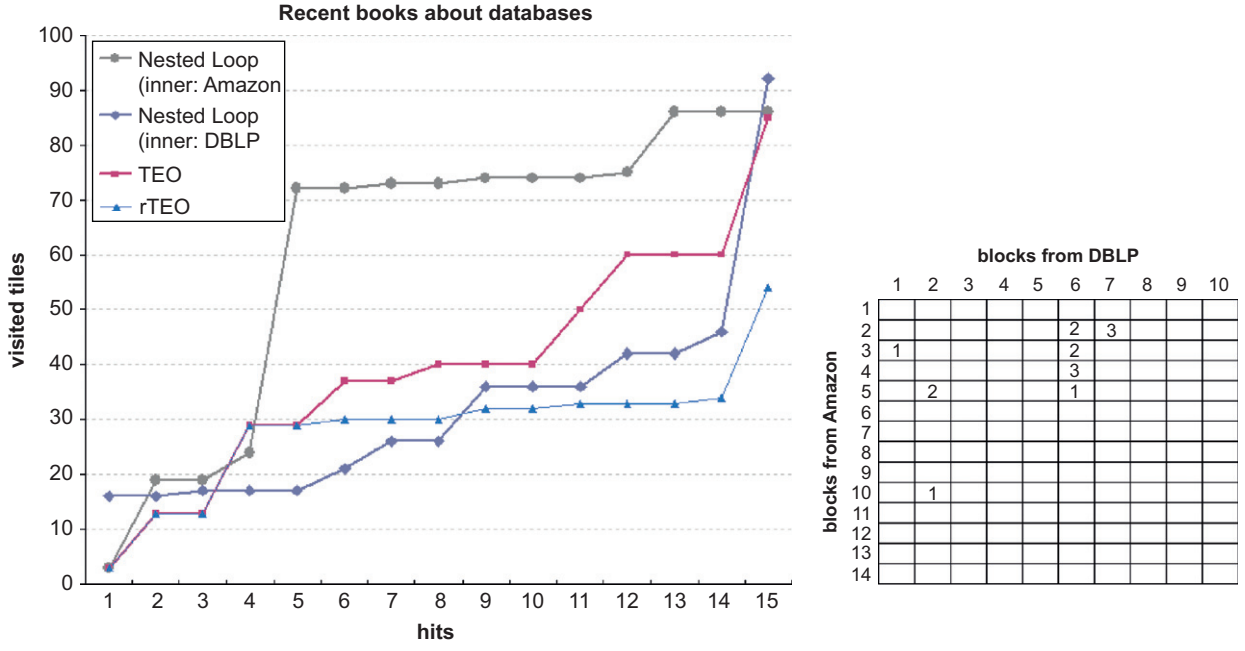

Fig. 14. Recent books about databases by active VLDB authors.

centered in a small area far from the origin of the axes, and in order to better approximate optimality one should detect such "locally promising areas" and heuristically try to maintain one of the parameters (a single element or a batch of elements, depending on the cost model) fixed for a while, so as to try to match it with all the results obtained from the other service—or at least to keep trying until the

"local enhancement" seems to have terminated. In order to take advantage of such opportunities, our framework can be extended as follows.

Generally speaking, extraction-optimality builds on the assumption that, given two items x_i and y_j and their combination r_k, the product of their ranking scores $\rho_i \times \rho_j$ is a good estimate of the *expected relevance* ρ'_k of r_k, as stated in Section 2.3. This is true only when the correlation of ρ_k and ρ'_k is strong. Often, the correlation between these variables is weak, so it happens that the expected relevance ρ'_k is different from the *actual relevance* ρ_k. For instance, Fig. 15 compares the trends of the expected and actual relevances for increasing values of the visited tiles, as observed in a given experiment. The figure illustrates several fluctuations in the actual relevance, while the expected relevance decreases linearly. These fluctuations are due to items with the same characteristics described in the previous example.

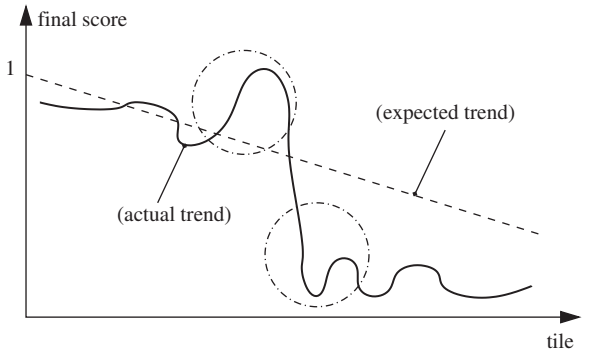

Fig. 15. Comparison of actual vs expected scores.

4.4. Runtime heuristics and adjustments

In this section, we sketch a heuristics that changes a given navigation strategy dynamically, by taking into account the actual ρ_k of the combined results returned by the matching operations. Actual ρ_k values can sometimes be computed, because an exact test can be done about the "matching" of each result entry with the query; in other cases, they can easily be specified by users, who empirically judge whether an entry belongs to the query result (e.g., by clicking on predefined options in the user interface); coherently, run-time heuristics may take advantage of the user's intervention.

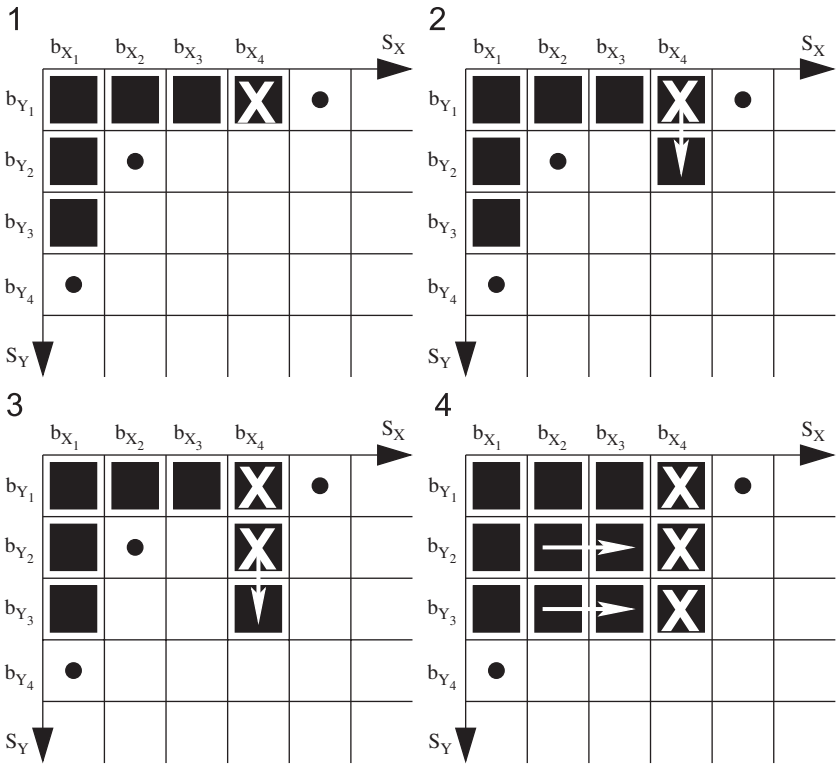

Fig. 16. Local greedy strategy.

In order to produce results more efficiently, several heuristic strategies can be used to try to take advantage of the *positive* fluctuations. When such a fluctuation is observed, a *greedy algorithm* explores the neighborhood of the last tile, where better results are likely to be found, disregarding the tile that would be chosen by the "static" algorithm.

An example of greedy heuristic is described in Fig. 16, as a variant of the TEO algorithm. Assume that a positive fluctuation is observed in the tile marked with a white X (1); then, the exploration of the tile immediately below X is suggested (2), and this strategy continues, while the search produces positive results, until the border of the search space (3), where the next tile would cause a request-response. At this stage, another heuristic could suggest completing the exploration of the (rectangular) search space (4); then, another heuristic would suggest continuing the search with the service S_X.[12] We have implemented a greedy heuristic that, in presence of positive fluctuations, modifies the TEO algorithm as indicated in Fig. 16, steps (1–4), and then resumes the static algorithm.

Fig. 14 shows the improved performance of the heuristic algorithm over the standard TEO (rTEO stands for "reinforced TEO").

5. Related works

Enhancing search engines is a known research focus since a long time, well represented in the Asilomar report [15]; a summary of hard Web search problems is in [16]. We briefly overview those research fields which constitute the premises to our work: search engines, information retrieval, web services, XML query languages, and object similarity measures.

5.1. Search engines and information retrieval

Search engines, among the most sophisticated and useful resources available on the Internet, assist the user in the task of rapidly and effectively navigating the web. To some extent, the problem of finding information on the Web can be rephrased as the problem of knowing where search engines are, what they are designed to retrieve, and how to use them. Two different types of engines have been developed so far: large-scale and specific search engines. Large-scale engines exemplify the trade-off between breadth and quality, while the specific ones are more likely to quickly focus a search in one particular area.

Historically, web search engines were born as applications of information retrieval techniques. Information retrieval systems are software tools which help users in the task of finding documents contained in a specific corpus or database. The most popular information retrieval technique involves combining the full text of all documents within a corpus into an inverted index. Search engines use the common tf-idf ranking algorithm (term frequency times inverse document frequency), to exploit basic statistical characteristics of natural-language so as to perform accurate retrieval [17]. Such systems are also widely used on the web for finding scholarly information as well as for many other recreational activities.

There have been few studies comparing the retrieval results of different search engines using different query formulations [18–20]. In [18] the authors present comparisons based on unrelated queries. Lucas and Topi [21] use eight search topics from which naive and expert queries were formulated and submitted to various web search engines to evaluate relevancy. Eastman [19] explores the precision of search engines using a variety of topics and query formulations, noting that precision did not necessarily improve with the use of the advanced query operators.

Ranking queries, also known as top-k queries [22,23] produce results that are ordered on some computed score. Typically, these queries involve joins, where users are usually interested only in the top-k join results. Top-k queries are dominant in many emerging applications, e.g., multimedia retrieval by content, web databases, data mining, middleware, and most information retrieval applications.

In [11] Bruno et al. propose an algorithm designed to minimize the number of object accesses, the computational cost, and the memory requirements of top-k search with monotone aggregate functions. In [24] Natsev et al. study the join of ranked results. Ilyas et al. [12] introduce a rank-join algorithm that makes use of the individual orders of its inputs to produce join results ordered on a user-specified scoring function. The idea is to rank the join results progressively during the join operation.

[12] These search strategies resemble the tactics of sailers during a race, when the race zone is explored looking for the "good wind", and this can come "from the right" (S_X) or "from the left" (S_Y); every positive fluctuation is a blow of the wind.

Two physical query operators are introduced based on variants of ripple join that implement the rank-join algorithm. The work proposes efficient heuristics designed to optimize a top-k join query by choosing the best join order. Better results are achieved by [25] exploiting properties of ranked results demonstrated by authors. An alternative approach to find top k results is offered by MystiQ [26], a system that uses probabilistic query semantics to find answers in large numbers of data sources of less than perfect quality. There are many reasons why the data originating from many different sources may be of poor quality, and therefore difficult to query: the same data item may have different representation in different sources; the schema alignments needed by a query system are imperfect and noisy; different sources may contain contradictory information, and, in particular, their combined data may violate some global integrity constraints; fuzzy matches between objects from different sources may return false positives or negatives. MystiQ relies on a probabilistic query semantics, and returns, along with each answer, its probability of matching with what the user really wants. It ranks the answers by this probability and returns them to the user.

A known approach to answering queries which pursue optimality with respect to more than one criterion is that of skyline queries [27]. The skyline of a set of d-dimensional points is the locus of the points that are not dominated by any other point on all dimensions. A point dominates another point if it is as good or better in all dimensions and better in at least one dimension. Therefore, the skyline of a query asking to maximize some quantities x and y is the set of points (each point representing a possible result) for which no other solution is better on at least one of x or y. This definition of dominance recalls that of our paper. Skyline computation has recently received considerable attention in the database community, especially for progressive (or online) algorithms that can quickly return the first skyline points without having to read the entire data file. Currently, the most efficient algorithm is NN (nearest neighbors) [28], which applies the divide-and-conquer framework on datasets indexed by R-trees. Although NN has some desirable features (such as high speed for returning the initial skyline points, applicability to arbitrary data distributions and dimensions), it also presents several inherent disadvantages (need for duplicate elimination if applied to more than two dimensions, multiple accesses of the same node, large space overhead). In [29,30] Papadias et al. propose BBS (Branch-and-Bound Skyline). It is a progressive algorithm also based on NN search, which is IO optimal, i.e., it performs a single access only to those R-tree nodes that may contain skyline points. Furthermore, it does not retrieve duplicates and its space overhead is significantly smaller than that of NN.

5.2. Web services

The basic language for describing web services is WSDL [31]. The composition of web services to constitute complex conversations is not supported by WSDL, but is the main purpose of several standardization proposals, logically positioned at the top of the so-called web service standard stack, the most recent of which are BPEL4WS [32], WSCI [33], and WSCL [34]. An example of formalization of web services is offered by [35]; the article describes a framework defining a set of elementary actions, and a set of web services whose implementation rely only on these actions.

Srivastava et al. introduced in [36] the concept of web service management system (WSMS), a system that enables querying multiple web services in a transparent and integrated fashion. The problem has several similar to the problem tackled in this paper. More specifically, an algorithm is proposed, which arranges web service calls into a pipelined execution plan which minimizes the total running time of the query by exploiting possible parallelism among web services and reducing the size of the data elaborated by the most costly services by anticipating the execution of those (less costly) services which in average return less output data than the data they take as input. The algorithm also determines the optimal granularity of data "chunks" to be used in input to each service call. The assumptions on the nature of web services and queries are quite strong (e.g., input attributes get their values from either exactly one other web service of from the user's input; recursive query plans are not allowed), but the described experiments show how an intelligent arrangement of service calls may significantly improve the overall performance. The model of WSMS proposed in [36] does not take into account on the peculiarities of *search* services, which are addressed in this paper.

Hull et al. give a broader view on the web services scenario [37], by using contributions from the theory of computation to asses the web service

properties that can be automatically inferred. Li and Horrocks describe in [38] the design and implementation of a service matchmaking prototype which uses a DAML-S based ontology and a description logic reasoner to compare ontology based service descriptions. Paolucci et al. [39] describe a technique for the semantic matching of web services. It has also been proposed (e.g., in [39]) to manually annotate web services, so as to enrich them with semantic information, to help services composition [40]. Heß and Kushmerick [41] study the problems of classification and clustering of services, with a supervised and unsupervised approach, respectively.

Fagin et al. [42] discuss the data exchange problem as that of starting from a source schema and creating an instance of a target schema which reflects the source data as accurately as possible. In such context they define an algorithm that merges data from different sources in polynomial time, under some strong hypotheses about the monotonicity of data returned by different sources with respect to the final ranking function. Despite the applicability hypothesis, which are too restrictive to apply directly the results to our context, the approach is interesting, as it shows that a simplified problem is indeed efficiently tractable even for large inputs.

This paper discusses the integration of search services so as to answer complex queries by leveraging simple search services already available. More long-term research concerns web service composition (i.e., the ability to combine web services which are capable of performing tasks so as to build complex workflows) and semantic web services [9,10] (i.e., the ability to select the services that best match a user's goal where both the goal and the services are semantically described by means of knowledge bases or ontologies).

Integration of heterogeneous data sources is gaining interest in the so-called "Web 2.0" community, as revealed by the growing excitement over the technique of mashups, which represent a new form of loosely coupled distributed system where content comes from different remote resources, while great emphasis is given to the separation from the presentation layer. Inspired by the work presented in [36] and by web mashups, [43] introduces the idea of continuous query over service-provided data feeds (e.g., through RSS or ATOM). The goal is to mash up and monitor the evolution of third-party feeds and to query the obtained result. Their mashup query model is articulated into collections of data items and collection-based streams of data (streams also track the temporal aspect of collections and allow the querying of the history of collections); suitable select, join, map, and sort operators are provided for the two constructs. The described system consists of a visual mashup composer, an execution engine, and interfaces for users to subscribe to mashup feeds equipped with personalized queries.

5.3. Textual queries over XML data

Since the introduction of XML, several textual query languages were proposed and analyzed by the database community [44,45], far before the proposal of XQuery [46]. One of the key benefits of XML is its ability to represent a combination of structured and unstructured (text) data. One can already find many real XML data repositories that contain such a mix of structured and text data such as the IEEE INEX data collection, ACM SIGMOD Record, Shakespeare's plays, DBLP, and Library of Congress documents. However, current XML query languages such as XPath and XQuery can only express basic queries over text data. An equally compelling paradigm for querying XML documents is full-text search on textual content. Research is ongoing to try to integrate these two querying paradigms.

TeXQuery [47] is a powerful full-text search extension to XQuery, which provides a rich set of fully composable full-text search primitives, such as Boolean connectives, phrase matching, proximity distance, stemming, and thesauri. TeXQuery also enables users to seamlessly query over both structured and full-text data by embedding TeXQuery primitives in XQuery, and vice versa. Finally, TeXQuery supports a flexible scoring construct that can be used to score query results based on full-text predicates. TeXQuery is also the precursor of the XQuery 1.0 and XPath 2.0 full-text language currently being developed by the W3C full-text task force (FTTF).

5.4. Similarity of data and services

The notion of similarity between objects finds use in many contexts, not only in search engines but also in collaborative filtering and clustering. Objects being compared are often modeled as sets, with their similarity determined based on set intersection. Intersection-based measures do not accurately

capture similarity in domains where data is sparse or characterized by known relationships between items within sets. Ganesan et al. propose in [48] new measures that exploit a hierarchical domain structure in order to produce more intuitive similarity scores.

Text similarity techniques are hardly applicable "as is" in the web service context, as textual documentation for invocable operations is succinct and unaware of structure information, which would be precious for conjecturing the semantics such operations. Indeed, input and output parameters of web service invocations may be strongly structured data items, provided with their proper schema specifications (WSDL). The issue of automatically matching different schemas has been widely addressed by the database community [49–51]. The body of research work in this field has developed various methods to collect information helping to characterize the semantics of data schemas, and also to guess and suggest suitable matching strategies based on such characterizations. These methods consider, among others, structural analysis, linguistic analysis, detection of synonymy, and semantic similarity, exploitation of domain knowledge, feedback from coupling similar schemas or coupling the same schema with other targets. Once more, nevertheless, the task of searching similar web service operations differs from that of general schema matching in some peculiar aspects: the search occurs at a different granularity, as matching operations is more similar to finding a similar schema than to looking for similar components in two given schemas (which, by the way, are already assumed to be in some mutual relationship); also, the operations of a single web service are usually not as strictly related one to another as the tables of a schema, and each web service per se does not provide much information as a database schema. Therefore, schema matching techniques are not ready to be adopted "as is" in our context.

6. Conclusions

In this paper, we have studied the problem of joining the results of heterogeneous search engines. In particular, we (i) formalized the problem in a simple but effective way; (ii) provided several examples of multi-domain queries whose automatic execution is hardly feasible by means of standard search technologies, that we nevertheless address with our approach; (iii) provided a reference architecture, a cost model and several execution strategies and optimization opportunities to guide the execution of multi-domain queries; (iv) described some experiments conducted with a prototype implementation.

Our prototype has been able to provide answers to multi-dimensional queries involving different and unrelated information sources. While the current prototype covers the design and implementation of join methods for search engines, we will next complete the development of the search engine integration framework, by developing the missing software modules. We also plan to define new join methods, based either on more sophisticated heuristics or on the interplay with human interaction.

References

[1] D. Beneventano, S. Bergamaschi, S. Castano, V. De Antonellis, A. Ferrara, F. Guerra, F. Mandreoli, G.C. Ornetti, M. Vincini, Semantic integration and query optimization of heterogeneous data sources, in: OOIS Workshops, 2002, pp. 154–165.
[2] D. Calvanese, G. De Giacomo, M. Lenzerini, M.Y. Vardi, View-based query processing: on the relationship between rewriting, answering and losslessness, Theor. Comput. Sci. 371 (3) (2007) 169–182.
[3] Z.G. Ives, A.Y. Halevy, D.S. Weld, Adapting to source properties in processing data integration queries, in: SIGMOD Conference, 2004, pp. 395–406.
[4] J. Madhavan, P.A. Bernstein, A. Doan, A.Y. Halevy, Corpus-based schema matching, in: ICDE, 2005, pp. 57–68.
[5] J. Madhavan, S. Cohen, X.L. Dong, A.Y. Halevy, S.R. Jeffery, D. Ko, C. Yu, Web-scale data integration: you can afford to pay as you go, in: CIDR, 2007, pp. 342–350.
[6] R. Pottinger, P.A. Bernstein, Creating a mediated schema based on initial correspondences, IEEE Data Eng. Bull. 25 (3) (2002) 26–31.
[7] R. Baumgartner, S. Flesca, G. Gottlob, Visual web information extraction with lixto, in: VLDB, 2001, pp. 119–128.
[8] V. Crescenzi, G. Mecca, P. Merialdo, Automatic web information extraction in the roadrunner system, in: Proceedings of ER (Workshops), 2001, pp. 264–277.
[9] WSMO, Web service modeling ontology ⟨http://www.wsmo.org/⟩.
[10] OWL-S ⟨http://www.daml.org/services/owl-s/⟩.
[11] N. Bruno, S. Chaudhuri, L. Gravano, Top-k selection queries over relational databases: mapping strategies and performance evaluation, ACM Trans. Database Syst. 27 (2) (2002) 153–187.
[12] I.F. Ilyas, W.G. Aref, A.K. Elmagarmid, Supporting top-k join queries in relational databases, VLDB J. 13 (3) (2004) 207–221.
[13] M. Arenas, P. Barcelo, R. Fagin, L. Libkin, Locally consistent transformations and query answering in data exchange, in: PODS, 2004, pp. 229–240.
[14] D. Beneventano, S. Bergamaschi, S. Castano, A. Corni, R. Guidetti, G. Malvezzi, M. Melchiori, M. Vincini,

Information integration: the MOMIS project demonstration, VLDB J. (2000) 611–614.
[15] P. Bernstein, M. Brodie, S. Ceri, M. Franklin D. DeWitt, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, J. Ullman, The asilomar report on database research, 1998.
[16] G. Weikum, J. Graupmann, R. Schenkel, M. Theobald, Towards a statistically semantic web, in: ER, 2004.
[17] I.H. Witten, A. Moffat, T.C. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
[18] V.N. Gudivada, V.V. Raghavan, W.I. Grosky, R. Kasanagottu, Information retrieval on the world wide web, IEEE Internet Comput. 1 (5) (1997) 58–68.
[19] C.M. Eastman, 30,000 hits may be better than 300, ASIST 53 (11) (2002) 879–882.
[20] B.J. Jansen, The effect of query complexity on web searching results, Inf. Res. 6 (1) (2000).
[21] W.T. Lucas, H. Topi, Form and function: the impact of query term and operator usage on web search results, JASIST 53 (2) (2002) 95–108.
[22] R. Fagin, Combining fuzzy information from multiple systems (extended abstract), in: PODS '96: Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM, New York, NY, USA, 1996, pp. 216–226.
[23] M.J. Carey, D. Kossmann, On saying "Enough already!" in SQL, 1997, pp. 219–230.
[24] A. Natsev, Y.-C. Chang, J.R. Smith, C.-S. Li, J.S. Vitter, Supporting incremental join queries on ranked inputs, in: VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, pp. 281–290.
[25] N. Mamoulis, M.L. Yiu, K.H. Cheng, D.W. Cheung, Efficient top-k aggregation of ranked inputs, ACM Trans. Database Syst. 32 (3) (2007) 19.
[26] J. Boulos, N.N. Dalvi, B. Mandhani, S. Mathur, C. Re, D. Suciu, Mystiq: a system for finding more answers by using probabilities, in: SIGMOD Conference, 2005, pp. 891–893.
[27] S. Borzsonyi, K. Stocker, D. Kossmann, The skyline operator, icde, 00:0421, 2001.
[28] D. Kossmann, F. Ramsak, S. Rost, Shooting stars in the sky: an online algorithm for skyline queries, in: VLDB '2002: Proceedings of the 28th International Conference on Very Large Data Bases, VLDB Endowment, 2002, pp. 275–286.
[29] D. Papadias, Y. Tao, G. Fu, B. Seeger, An optimal and progressive algorithm for skyline queries, in: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2003, pp. 467–478.
[30] D. Papadias, Y. Tao, G. Fu, B. Seeger, Progressive skyline computation in database systems, ACM Trans. Database Syst. 30 (1) (2005) 41–82.
[31] W3C, Web Services Description Language (WSDL) ⟨http://www.w3.org/TR/wsdl⟩, March 2001.
[32] Business Process Execution Language (BPEL) ⟨http://www-128.ibm.com/developerworks/⟩, ⟨library/specification/ws-bpel⟩, July 2002.
[33] W3C, Web Service Choreography Interface (WSCI) ⟨http://www.w3.org/TR/wsci⟩, August 2002.
[34] W3C, Web Services Conversation Language WSCL ⟨http://www.w3.org/TR/2002/NOTE-wscl10-20020314⟩, March 2002.
[35] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, E-service composition by description logics based reasoning, in: Description Logics, 2003.
[36] U. Srivastava, K. Munagala, J. Widom, R. Motwani, Query optimization over web services, in: VLDB-06, 2006, pp. 355–366.
[37] R. Hull, M. Benedikt, V. Christophides, J. Su, E-services: a look behind the curtain, in: PODS, 2003, pp. 1–14.
[38] L. Li, I. Horrocks, A software framework for matchmaking based on semantic web technology, in: WWW '03: Proceedings of the 12th International Conference on World Wide Web, ACM Press, New York, NY, USA, 2003, pp. 331–339.
[39] M. Paolucci, T. Kawamura, T.R. Payne, K.P. Sycara, Semantic matching of web services capabilities, in: ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web, Springer, London, UK, 2002, pp. 333–347.
[40] J. Bézivin, J. Hu, Z. Tari (Eds.), Web services: modeling, architecture and infrastructure, in: Proceedings of the 1st Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI-2003), In conjunction with ICEIS 2003, Angers, France, April 2003, ICEIS Press, 2003.
[41] A. Heß, N. Kushmerick, Learning to attach semantic metadata to web services, in: International Semantic Web Conference, 2003, pp. 258–273.
[42] R. Fagin, P.G. Kolaitis, L. Popa, Data exchange: getting to the core, ACM Trans. Database Syst. 30 (1) (2005) 174–210.
[43] J. Tatemura, A. Sawires, O. Po, S. Chen, K. Selcuk Candan, D. Agrawal, M. Goveas, Mashup feeds continuous queries over web services, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, New York, NY, USA, 2007, pp. 1128–1130.
[44] M. Fernández, J. Siméon, P. Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, J. Widom, Xml query languages: experiences and exemplars, 1999. Available from ⟨http://www-db.research.belllabs.com/user/simeon/xquery.ps⟩.
[45] Z.G. Ives, Y. Lu, Xml query languages in practice: an evaluation, in: Proc. WAIM'00, 2000, pp. 29–40.
[46] W3C, XQuery: an XML query language, ⟨http://www.w3.org/XML/Query⟩, November 2003.
[47] XQuery 1.0 and XPath 2.0 Full-Text ⟨http://www.w3.org/TR/xquery-full-text⟩.
[48] P. Ganesan, H.G. Molina, J. Widom, Exploiting hierarchical domain structure to compute similarity, ACM Trans. Inf. Syst. 21 (1) (2003) 64–93.
[49] E. Rahm, P.A. Bernstein, A survey of approaches to automatic schema matching, VLDB J Very Large Data Bases 10 (4) (2001) 334–350.
[50] H.H. Do, E. Rahm, Coma—a system for flexible combination of schema matching approaches, in: VLDB, 2002, pp. 610–621.
[51] A. Doan, P. Domingos, A.Y. Halevy, Reconciling schemas of disparate data sources: a machine-learning approach, in: SIGMOD Conference, 2001, pp. 509–520.